2013 IEEE 27th International Conference on Advanced Information Networking and Applications

An Efficient TCAM Update Scheme for Packet Classification

Yeim-Kuan Chang
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan
ykchang@mail.ncku.edu.tw

Kai-Yang Liu
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan
p76001433@mail.ncku.edu.tw

Abstract—**Ternary Content Address Memory (TCAM) becomes a popular hardware device for storing the packet classifiers due to the advantages of high and deterministic lookup performance. However, managing the filter set in TCAM is quite complicated when filters need to be updated. To ensure the correctness of search results, it is required to obtain the right position in TCAM for storing the new filter. In addition, some filters need to be moved to other positions for maintaining the correct filter overlapping relationship based on their priorities. Normally, an auxiliary data structure is used to compute how to insert or delete a filter into/from TCAM. However, this auxiliary data structure is usually complicated and large. Instead of maintaining an auxiliary data structure, in this paper, we propose an efficient TCAM update scheme that simply uses a very small portion of TCAM search cycles to compute how to move the filters related to the filter to be inserted or deleted. Therefore, a large amount of memory for storing the auxiliary data structure along with the local CPU for updates can be avoided. In addition, our simulation results show that the proposed update scheme needs less number of TCAM movements than the existing CoPTUA update scheme [2].**

Keywords- packet classification, TCAM , rule update

I. INTRODUCTION

Packet classification is an important mechanism employed by internet routers. The packet header consists of five fields: source address, destination address, source port, destination port, and protocol. To classify packets, the values of these fields are extracted to match a list of rules. A rule can be regarded as (***F***, ***A***), where ***F*** is a filter composed of the five fields, and ***A*** is an action that will be taken when the filter is matched by the packet header fields. If multiple filters are matched, then the action of the rule with the highest priority will be taken.

Storing filters in Ternary Content Address Memory (TCAM) is a popular hardware-based solution for packet classification. TCAM-based packet classification stores each filter in a TCAM entry in ternary format (i.e. each cell can be one of the three states: 0, 1, and ***x*** (don't care)) and all the filters are stored with decreasing priority order. The advantage of TCAM-based packet classification is that the lookup operation only takes one cycle. When a packet arrives, all the entries in the TCAM are searched in parallel to find the matched filter with the highest priority. Thus, TCAM provides high and deterministic search performance for packet classification. However, TCAM-based packet classification has some well-known disadvantages:

- *High Power Consumption*:

Performing parallel search on all entries to classify a packet will result in high power consumption due to the priority encoder and parallel search circuits in TCAM chips. Therefore, the power efficiency will degrade further when the number of filters in the TCAM increases. The drawback of high power consumption can be regarded as a tradeoff to achieve high search performance. To address this problem, some approaches have been proposed to group the TCAM entries [9][12][13], and these groups can be enabled or disabled selectively when searching the TCAMs. Thus, the power efficiency improves if the TCAM entries are grouped properly.

- *Range Expansion Problem*:

In TCAM-based packet classification, the obstacle of storage inefficiency occurs due to the existence of ranges in some fields of filters. For example, source port and destination port are numbers represented in the form of ranges. One arbitrary range is not allowed to be stored directly in a TCAM entry if the range can't be represented as a prefix. The traditional solution is to convert each range into multiple prefixes by using the direct range to prefix conversion algorithm [10]. However, using multiple TCAM entries to represent a rule will results in the problem of storage inefficiency and higher power consumption. Many approaches have been proposed to improve the storage requirement for this issue [4][16][17].

Efficiently updating the filter sets is another issue for TCAM-based packet classification. The works that focus on this issue are fewer than the others. During the update period, some new filters may be inserted and some existing filters may be removed. Managing filters in the TCAM becomes complicated since the new filters have to be inserted at the right position according to their priorities. In addition, some filters need to be moved to ensure the correctness of lookup results. Generally, an auxiliary data structure such as priority graph is maintained to record the priority relationship among filters and the TCAM entry positions of filters. Therefore, we can know how to perform the TCAM update process by the auxiliary data structure. The update process of packet classifiers is normally

1550-445X/13 $26.00 © 2013 IEEE
DOI 10.1109/AINA.2013.96

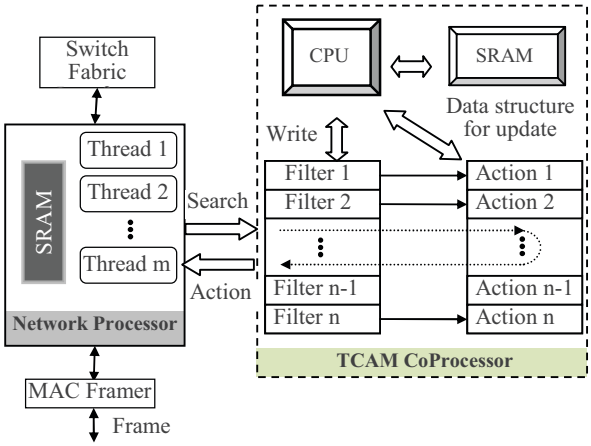


Figure 1(a) A conventional TCAM coprocessor/network processor.

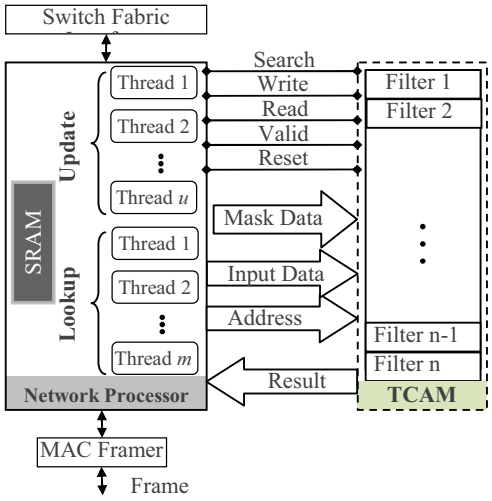


Figure 1(b) The proposed TCAM coprocessor/network processor.

performed by a local CPU/ TCAM coprocessor interface (e.g., the 64-bit PCI bus in the INTEL IXP2800 network processor [18]). The architecture of conventional TCAM coprocessor with a network processor used for packet classification is shown in figure 1(a). The filters which map to distinct memory addresses in an associated SRAM are stored in the coprocessor, and the corresponding actions are stored in the SRAM. The search and update requests are issued by the lookup threads of the network processor and the local CPU separately. During the update process, the inconsistent and erroneous results can be avoided by locking the interface. However, locking the interface during the update process leads to the reduction of search performance. Note that we don't discuss the dual-ported TCAM which is accessible from a local CPU and a network processor concurrently. Thus, the algorithm for maintaining the consistency of the filter set which usually has high update effort is not required in this paper. The auxiliary data structure mentioned before usually stored in off-chip SRAM is also shown in figure 1(a). Unfortunately, a large amount of memory is usually required to store the auxiliary data structure such as priority graph.

In this paper, we propose an efficient TCAM update scheme for packet classifiers. We purely use TCAM search operations to compute how to move TCAM entries. Therefore, our proposed update scheme does not need a large amount of memory for auxiliary data structure that would usually be stored in off-chip memory. Therefore, the off-chip memory and local CPU are not needed in the TCAM coprocessor. The proposed TCAM update architecture is shown in figure 1(b). The TCAM needs to be designed to support multiple matching in our algorithm. Many approaches have been proposed for supporting multiple matching, and we will discuss further in section 2.

The rest of the paper is organized as follows: We review the related works in Section II. We describe our update scheme in Section III. The experimental results of our scheme are shown in Section IV and we conclude in Section V.

II. RELATED WORKS

In this paper, we focus on the problem of managing the filter sets when updating packet classifiers in the TCAM. In this section, we first introduce other existing TCAM update scheme for packet classifier. Since our TCAM needs to be designed to support multiple matching, we also briefly describe the background of TCAM multiple matching.

A. Approaches for updating packet classifiers

Different approaches have been proposed to manage the filters in the TCAM for updating packet classifiers [1][2][3][14][15]. The authors in [5] proposed two well-known algorithms, PLO_OPT and CAO_OPT for updating prefixes. Both algorithms significantly reduce the number of TCAM entry moves in the worst case by keeping the empty entries in the center of the TCAM. However, these two algorithms can't be applied to updating packet classifier rules directly. Filters can also be grouped and represented as chains by the way similar to CAO_OPT algorithm for prefixes according to their overlapping and priority relationship. However, compared with updating prefixes, there are more constraints need to be considered when updating filters. The difference is that inserting a filter may lead to merging multiple distinct chains, which will probably increase the number of TCAM entry moves and the complexity of computation. In addition, the lengths of prefix chains must be less than or equal to the width of ip address field (e.g., 32 in IPv4). However, the lengths of filter chains may be much longer than that of prefix chains since the lengths of filter chain do not have upper bound.

The authors in [2] proposed a consistent algorithm to update a batch of rules. This algorithm focuses on how to maintain the consistency of filter set in order to avoid locking the TCAM coprocessor during the update process. Therefore, rule matching and updating can be processed simultaneously by using the dual-ported TCAM, and the correctness of rule matching is ensured even in the progress of update. In order to

maintain consistent filter set for each step of moves, this algorithm needs several moves to complete updating a batch of rules. Although the TCAM can be searched during update progress, a lot of TCAM bandwidth is still wasted for moving the entries. In addition, it may take a long time for new rules to wait for the completion of previous batch to get in the update progress.

The authors in [1] proposed a fast way to complete updating rules by inserting the new filters into arbitrary entries without considering the order of priorities among overlapped filters. To get the matching results correctly, an auxiliary graph is used to maintain the priorities and overlapping relationship among these filters. Each filter is given a priority value which is equal to one plus the maximum priority value among all the filters that overlap it and have higher priority, one is given if no such filter exists. The priority values are further appended to each filter and stored in the extra bits of TCAM entries. The correct lookup results are obtained by searching the matched filter with the minimum priority value instead of minimum index. However, this approach will degrade the efficiency of lookup procedure since it requires $\log_2 N$ lookups to determine the filter with highest priority by using binary decision tree, where N is the distinct priority values.

PC-DUOS scheme proposed by [3] is extended from DUOS [11]. Two TCAMs called LTCAM and ITCAM are used to store filters in PC-DUOS. The LTCAM is used to store the filters with highest priority among all overlapped filters and all filters in the LTCAM are disjoint. Therefore, the priority encoder is not required in the LTCAM. The remaining filters are stored in the ITCAM and the priority graph proposed by [1] is needed to manage the filters in ITCAM to ensure the correctness of lookup results. A multi-dimensional trie is maintained to determine which filters should be stored in the LTCAM. LTCAM and ITCAM are searched in parallel during lookup. The search result of ITCAM is discarded if there is a match found in LTCAM since the matched filter in the LTCAM must have higher priority than the one in the ITCAM. To update a filter, the multi-dimensional trie is first updated and used to compute the moves for managing the filter set. However, the priority graph is also needed to be updated when rearranging the filters in the ITCAM.

B. TCAM multiple matching

A lot of different approaches have been proposed to solve the TCAM multiple match classification problem [4][6][8]. There is an additional valid bit for each TCAM entry to indicate whether the entry is active or not. Only the active entries are involved in search process. To address the problem of multiple matching, the simplest way is modifying the valid bit of the best matched entry. This ensures that the entry will not participate in the search process of next cycle. Thus, the second match is reported after the next lookup by using the same search key. The process repeats until all the matches are found. The valid bits of all matched entries need to be set to valid again for the next search key. However, write operations are needed to modify valid bits. It takes 7 cycles per multi-match (3 cycles for write and 1 cycle for search operation) as mentioned in [4].

The authors in [4] proposed an algorithm that uses the extra bits to store the index number of that entry called discriminator field. The search key is appended with the discriminators when performing TCAM search operation, and the discriminators are set to don't care initially. After reporting the first match at position ***j***, the discriminators are set to prefixes for range greater than ***j*** to get the second match. This can be done by utilizing the capability of global masking.

In order to accomplish the multiple matching by using only one TCAM search operation, geometric intersection-based approaches [8] creates additional entries to record the intersection filter. For example, if two filters that overlap with each other stored in the TCAM, a new intersection filter of these two filters is created and stored above them. Therefore, the intersection filter can be regarded as the multiple matching of these two filters. However, the problem of memory inefficiency and high power consumption occurs when the overlapping relationship is severe in the filter set. To address this problem, [6] proposed a *set splitting algorithm (SSA)* to efficiently group the filters into multiple blocks. Thus, memory requirement and power consumption can be reduced significantly while just a little search time is sacrificed.

III. PROPOSED TCAM UPDATE SCHEME

The filters' indices in the filter sets can be treated as their priority values where the filter with lower index has higher priority. Therefore, we can store entire filter set in TCAM without any rearrangement and the correctness of lookup results is ensured. However, it may cause many TCAM entry moves when we need to insert a new filter. In the worst case, all filters need to be moved. To address this problem, we first create the priority graph proposed in [1] for the filters that initially exist. The filters are further grouped and rearranged according to the graph.

An example of priority graph is shown in figure 2. We use five 2-dimensional filters in our example and the regions of these filters are shown in the left part of figure 2. In the priority graph as shown in the middle part of figure 2, each vertex denoted by a circle represents a filter. A directed edge is created from vertex ***i*** to vertex ***j*** if filter ***i*** overlaps ***j*** and filter ***i*** has higher priority. Each filter is given a priority value which is equal to one plus the maximum priority value among all of its ancestors. If the filter has no ancestor, its priority is set to one. The priority values of all filters in this example are shown in the right part of figure 2.

We use the priority graph to group the filters into layers as follows. We divide TCAM into ***L*** layers where ***L*** is the maximum priority value of the priority graph. The filters with priority value ***i*** will be grouped into layer ***i*** for $i = 1$ to ***L***. In addition, the filters in layer ***i*** must be put above layer ***j*** for $i < j$ in TCAM. Note that the priority graph is no longer needed after the grouping procedure mentioned above. Some properties will always hold after grouping and we summarize

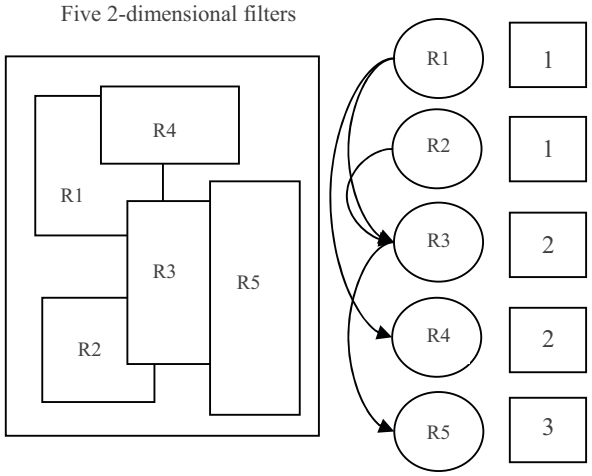


Figure 2: An Example of Priority Graph

these properties below:

1. The filters in the same layer are disjoint.
2. If there are multiple matches in more than one layer for a search key, the matched filter in layer ***i*** must have higher priority than that in layer ***j*** for ***i*** < ***j***.
3. A filter in layer ***i+1*** overlaps at least one filter in layer ***i***.

The empty TCAM entries have to be put in some place so that the update process can be done successfully. If we allocate empty TCAM entries for each layer, it is not necessary to move any entry after inserting a filter. However, it is not a good solution since we are not sure whether we have enough empty TCAM entries for all layers. In addition, if we allocate too many empty TCAM entries, the power consumption may be high. Therefore, where to place these empty entries in TCAM is an important issue. According to our experiment, a large portion of filters are in layer 1 and 2. Thus, it is reasonable to conjecture that the updated filters would have greater chance processed in these two layers, so we propose to place the empty TCAM entries between them. Figure 3 shows our TCAM configuration for storing the five filters in figure 2. We use the extra bits to encode the layer field and discriminator field for the capability of multiple matching, and we will explain in detail later.

In order to guarantee the correctness of lookup results, we have to ensure that the three properties mentioned before are still satisfied after a filter is inserted or deleted. To address the range expansion problem, we simply apply the *direct-range-to-prefix conversion* (DRPC) algorithm [10]. Subsequently, we introduce a specific algorithm to insert or delete a filter. That is, the source port and destination port of the filter we discuss can be represented by a prefix. Instead of using the auxiliary data structure, our algorithm purely uses TCAM search operations to determine the relevant filters (the filters that would be involved during the update procedure) for calculation. Only a little information needs to be maintained for managing the filter sets in our update scheme.

Index	Filter	Layer	Discriminator
1	R1	1	1
2	R2	1	2
	N empty entries		
N+3	R3	2	N+3
N+4	R4	2	N+4
N+5	R5	3	N+5

Figure 3: TCAM Configuration.

*A. Inserting filter **R***

The algorithm for insertion can be divided into two parts: *Calculation* and *TCAM entry movement*.

A.1 Calculation:

Before inserting the new filter, we need to calculate where to put the filter and which entries should be moved to other place. To be precise, if the new filter ***R*** overlaps some existing filters, we have to ensure that these filters still satisfy property 2 after insertion. Our scheme simply uses TCAM search operations to determine the filters that overlap filter ***R***. We can achieve this by making filter ***R*** as search key to perform TCAM search operation. Then the matched filters must overlap filter ***R***. However, conventional TCAM only output the filter with the highest priority, in order to find out all the overlapped filters, our TCAM should be designed to support multiple matches.

The approaches for supporting TCAM multiple matches are discussed in section 2. We apply MUD approach in our algorithm due to the advantage of lower power consumption compared with other schemes. In addition, it is convenient to disable the capability of multiple matching by masking the discriminator field when only the best match needs to be reported. As described in [4], we use the index number of TCAM entry to encode the discriminator field. In addition, we create a layer field for storing the layer number of each filter.

With the help of multiple matching approaches, we can easily determine the filters which overlap filter ***R***. That is, all of the matched filters are exactly the ones that need to be taken into consideration. If we consider these matched filters and ***R***, the relationship among them can be illustrated by using *Connected Rule Graph (CRG)* [2]. An example of *Connected Rule Graph* is shown in figure 4 (***C*** is assumed to be the new filter). The direction of the arrows implies the increase in priority, and a horizontal line represents the 1-dimensional range that the filter covers. For each filter in the same CRG, there is at least one filter that overlaps it.

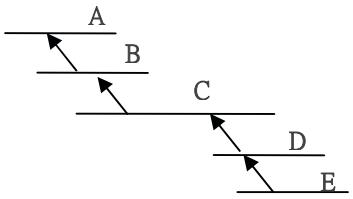


Figure 4: An Example of Connected Rule Graph

Now, our goal is to insert filter ***R*** into right position and maintain the priority relationship of whole CRG. Note that inserting a new filter may cause originally different CRGs to be merged into one. For example, before we insert filter ***C*** in figure 4, there are two CRGs where one includes filters ***A*** and ***B*** and the other includes filters ***D*** and ***E***. According to the grouping properties, filter ***D*** needs to be moved from layer 1 to 4, and filter ***E*** needs to be moved from layer 2 to 5 after inserting filter ***C***.

In our algorithm, the filter ***R*** is first used as search key to perform multiple matching with all TCAM entries. For each matched filter, the corresponding layer and TCAM entry index need to be recorded in buffer for calculation. In calculation phase, we first compare all the matched filters' priorities with ***R***. We accomplish this procedure by storing the filter index as priority value in the SRAM. Further, we can divide the matched filters into two groups: *Group 1* for filters that have higher priority than ***R***, and *Group 2* for those that have lower priority.

Now, we discuss where to insert filter ***R***. In order to satisfy property 2, we should ensure the matched filters that have higher priority than ***R*** to be placed in the layers higher than *R* (i.e. ***R*** has larger layer number after insertion). We first find the lowest layer, say *k*, at which one of the filters in *Group 1* locates. Then we insert filter ***R*** in layer *k+1*.

In *Group 2*, the filters in layer ≤ (*k+1*) should be moved to the layer below layer (*k+1*) to satisfy property 2. We move these filters in the increasing order of their layer numbers (i.e. selecting the filter with lowest layer number to move first). Consider a filter *X* in Group 2 where *X* is in layer *m* and $m \le k + 1$. We first delete *X* from layer *m*. Then we use the same insertion algorithm to insert *X* in one of the layers from *k*+2 to *L*+1, where *L* is the largest layer number before inserting *X*. We don't have to check the matched filters again in layers ≤ *m* since the matched filters in layers ≤ *m* do not affect the decision of re-inserting *X*. Thus, the procedure of multiple matching can start from layer *m+1* to reduce the time for calculation. In fact, it is not necessary to multiple-match all the TCAM entries in most cases. Assume *X* is overlapped with one of the filters in layer ***i*** during the procedure of multiple matching. If *X* has a priority higher than all the filters in layer ***i***, then the filters in layer > ***i*** do not affect the decision of where to insert *X* since it's impossible to find a matched filter against *X* that has a higher priority than *X* in layer > ***i***. For example, assume a filter *Y* in layer *i* + 1 is overlapped with *X*. Based on

```
//Insert filter R with priority P into the TCAM. L is the original
layer number of the filter that needs to be re-inserted. If R is the
new filter, then Insert(R, P, 0) is performed.

Insert(R, P, L)
{
01  Start = Start_index[L]; // initial index for multiple matching
02  While(1){ // multiple matching process
03  result.match = TCAMsearch(R, Start);// Search TCAM from
04                                              index Start
05   if (result.match =  0 )  break; // No match is found
06   else {
07            if(Match_Filter.Priority > P)
08               group1[Cnt1++] = Match_Filter;
09            else  group2[Cnt2++] = Match_Filter;
10
11           if (P > Layer_Max[Match_Filter.Layer])  break;
12    }  //end else
13      Start = match.index+1; //Match from Start for next cycle.
14  } //end while
15
16  Layer_insert = 1+Max{Layer number of group1};
17  Insert_TCAM(R, Layer_insert);
18  for( i = 0; i < Cnt2 ; i++){ //Cnt2 : number of filters in Group2
19       if(group2[i].Layer < = Layer_insert){
20  Insert(group2[i].Filter, group2[i].Priority, group2[i].Layer);
21       Delete group2[i].Filter by invalidating the entry.
22       }
23  } //end for
24  Clear all the elements stored in group1 and group2.
}
```

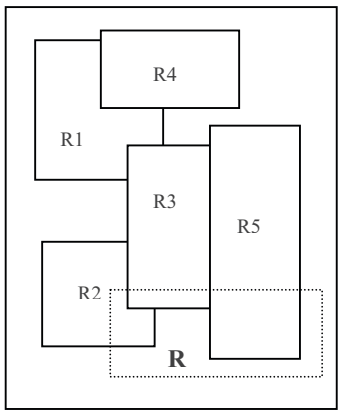
Figure 5(a). Filter insertion algorithm.

```
//Insert filter R into Layer L in the TCAM.

Insert_TCAM(R, L)
{
01   if (Freelist[L]!=NULL){
02      Remove the first free index (idx) from Freelist[L];
03      TCAMwrite(idx, L, R); // write filter R in the free index
04                                  (idx) which is in layer L.
05   }else{
06         Find the nearest layer u to layer L;
07         remove the first free index  (idx) from Freelist[u];
08          if(u > L){
09               for(i = u to L)
10                Move the uppermost filter of layer i to the free
11                entry to create a new free entry in layer (i-1).
12         } else{ // u < L
13                 for(i = u to L)
14                     Move the lowermost filter of layer i to the free
15                     entry to create a new free entry in layer (i+1).
16            }
17       TCAMwrite(idx, L, R);
18     } // end else
}
```

Figure 5(b). The algorithm for inserting a filter in layer *L*

property 3, Y must overlapped with another filter *W* in layer *i* and *W* has a priority higher than *Y*. Since *X* has a priority higher than all the filters in layer ***i***, we must know that the priority of *X* is higher than *Y*. As a result, filter *X* must be inserted in one of the layers from 1 to *i*. We use an array *Layer_max[i]* to record the highest priority in layer ***i***. If a



(a) Inserting filter ***R*** into filter set.

Valid	Filter	Layer	Discriminator
1	R1	1	1
1	R2	1	2
N-1 empty entries			
1	**R**	2	N+2
0			
1	R4	2	N+4
1	R5	3	N+5

(b) ***R3*** should be deleted and re-inserted after inserting ***R***

Valid	Filter	Layer	Discriminator
1	R1	1	1
1	R2	1	2
N-1 empty entries			
1	**R**	2	N+2
1	R4	2	N+3
1	R3	3	N+4
1	R5	4	N+5

(c) The final configuration after the rearrangement

Figure 6: The process of inserting a new filter in our algorithm

match occurs in layer ***i*** during the procedure of multiple matching, we compare *X's* priority with *Layer_max[i]* to decide whether the multiple matching procedure should be terminated or not. The *Layer_max[i]* array should be updated when the highest priority filter in layer ***i*** is moved to another layer.

A.2 TCAM entry movement:

For each layer ***i***, we use a linked list *Free**list[i]* to record the position of TCAM empty entries (the empty space for layer 1 and 2 can be regarded as infinite), and *Start_index[i]* to record the index of the first filter in layer ***i.*** Therefore, we can immediately insert the new filter ***R*** in one empty entry after we know which layer to store the filter ***R***. In this case, we don't need to move any entry to complete the insert operation. However, if there are no empty entry in layer ***i***, we need to find one empty entry from other layers for ***R***. Some boundary filters (the filters that stored in the first or last entry of the layer) have to be moved to create a free entry for ***R***.

To minimize the moves for this process, we propose to select the empty entry closet to the layer ***k*** (assume ***k*** is the layer into which ***R*** needs to be inserted). We can easily determine it by checking the *Freelist*. Assume the empty entry we selected is in layer ***e***, then $|e - k|$ boundary entries need to be moved to create an empty entry for the new filter. *Start_index* also needs to be updated when the position of the first filter in each layer changes.

Figure 5(a) shows the detailed algorithm *Insert*(*R*, *P*, *L*) to insert a new filter or re-insert an existing filter with priority *P*. This algorithm first determines the initial index for performing multiple matching. In the procedure of multiple matching, the required information of the matched filters such as layer number is stored in *group1* or *group2* according to their priorities compared with *P* as shown in line 7-10. Further, the steps for update are determined as shown in line 16-23. The algorithm for inserting a filter in layer *L* is also shown in figure 5(b).

B. Deleting filter F

In order to delete the filter ***F***, we should first determine if ***F*** exists in the TCAM. In addition, we need to know which entry does ***F*** stored in if ***F*** exists. To complete the above calculation efficiently, we utilize the idea of *MUD* algorithm again. Instead of using TCAM index of an entry as its discriminators, we create a new field called layer field and use $\lceil \log_2 (L+1) \rceil$ extra bits in each TCAM entry to record its corresponding layer number, where L is the maximum layer number. It is possible that many filters in the same layer match ***F***, but we only need to check the first matched filter and lookup its priority value to determine if the filter is the same with ***F***. The reason is that the filters in the same layer are disjoint (property 1). If any matched filter in layer ***i*** is not ***F***, then we can ensure that ***F*** is not in layer ***i***. Therefore, we can use layer number as discriminators and apply MUD algorithm to quickly determine the position of ***F***.

Note that deleting ***F*** may violate property 3. Assume that ***F*** is in layer ***L***, for the filters in layer (***L+1***) that only overlap ***F*** in layer ***L*** would not satisfy property 3 after the deletion of ***F***. We have to move these filters called ***F'*** to layer ***L***. In order to determine the filters that have to be moved, we use multiple matching scheme again to record the filters that overlap ***F*** in layer (***L+1***) in buffer. Then we delete ***F*** by invalidating the TCAM entry and record its position in *Freelist*[***L***] . For each filter in buffer, we use it as search key to match the filters in layer ***L***. The filter should be moved to Layer ***L*** if there is no match found. The same process should be recursively executed when we delete some ***F'*** in Layer (***L+1***).

We use the same filter set shown in figure 2 and 3 to illustrate our steps for insertion. If we want to insert filter ***R*** in figure 6(a), and ***R***'s priority is assumed to be higher than ***R3***'s and lower than ***R2***'s. We use filter ***R*** as search key to start multiple matching. Both layer and discriminator fields of search key are set to don't care initially. ***R2*** will be reported first since it is stored in the lowest memory location among the

filters that overlap ***R***. The layer number and TCAM entry index of ***R2*** are stored in buffer. The discriminator field of the search key is further modified to be a set of prefixes for representing range 3 to N+5 to continue the multiple matching. ***R3*** is the next matched filter and the same information is also stored in buffer. Since ***R***'s priority is greater than all filters' in layer 2 (the layer that ***R3*** is stored in), the multiple matching is terminated. ***R*** is inserted in layer 2 because there is only ***R2*** in layer 1 that overlaps ***R*** and has higher priority. We allocate an empty entry in layer 2 for storing filter ***R***. The filters with layer number ≤ ***R***'s in buffer that has lower priority have to be re-inserted. ***R3*** is deleted and start the process for re-insertion. Figure 6(b) shows the TCAM configuration after inserting ***R*** and deleting ***R3***. Note that we don't check the filters in layer 3 to determine which filters need to be moved to layer 2 immediately after deleting ***R3***. This is because that the filters in layer 3 that overlap ***R3*** may also have to be re-inserted later. The difference between re-insertion and insertion is that re-insertion can start the multiple matching from the filter's original layer. ***R3*** is determined to be stored in layer 3 and ***R5*** needs to be re-inserted after calculation of the re-insertion of ***R3***. The final TCAM configuration for inserting filter ***R*** is shown in figure 6(c).

IV. PERFORMANCE

We generate large synthetic rule sets by using the tool ClassBench [7] to evaluate the update performance of our scheme. There are three types of filter sets called access control list (ACL), firewall (FW), and IP chain (IPC). We generated 10K rules for each type in our simulation. We first randomly selected 90% filters from the entire filter set. These filters are initially rearranged in the TCAM according to the priority graph. The other 10% filters are treated as insertion trace. In addition, we also randomly marked 10% filters from the selected 90% filter subset as deletion trace. We alternatively performed inserting and deleting the filters in trace files in our simulation. The time required for updating the classifiers is the most important metric to evaluate the update performance. The update period can be divided into two parts: Calculating how to rearrange the filter set and completing the rearrangement by performing TCAM write operations. We will discuss these two parts in detail separately.

A. Calculation:

Generally, an auxiliary data structure is needed to maintain the overlapping relationship and the priority order among filters. When a new filter arrives, the auxiliary data structure needs to be updated first, and the right position for the new filter is obtained. Unfortunately, updating the auxiliary data structure in off-chip memory is time consuming. In addition, a large amount of memory is usually required to store the auxiliary data structure. For example, each vertex of the priority graph needs to maintain the overlapping relationship with all the other vertices. In addition, other information such as the TCAM entry index that the corresponding filter stored in also needs to be recorded. Our scheme purely uses TCAM search operation to complete the calculation phase instead of maintaining an auxiliary data structure. Table I and II show the experimental results of calculation time in terms of the number of TCAM search operations for insertion and deletion separately in our scheme. According to our experimental results, our proposed scheme only takes about tens to hundreds additional clock cycles per filter to substitute the update process for auxiliary data structure. We analyze the reason why firewall filter set has worse performance compared with the others. First of all, the range expansion problem of firewall is more severe than the others, which increases the chance to degrade the efficiency of MUD algorithm. Second, it takes more cycles to complete multiple matching since FW has more overlapping filters. However, it's obvious that updating the auxiliary data structure for FW also takes longer time since more filters are involved in the update process.

TABLE I. Calculation results for insertion

Filter Set	Avg # of lookups per filter	Max # of lookups per filter
acl_10K	5.54	506
fw_10K	269.88	9965
ipc_10K	38.28	1986

TABLE II. Calculation results for deletion

Filter Set	Avg # of lookups per filter	Max # of lookups per filter
acl_10K	2.30	84
fw_10K	16.64	1635
ipc_10K	6.65	264

B. Entry Rearrangement

The number of TCAM entry moves is the most common metric for evaluating the update performance. Moving an entry includes one TCAM write and one TCAM delete operation. Grouping the filters reduces the numbers of entry moves in the worst case. However, the update trace is closely related to the performance of rearrangement. Consider a new filter which overlaps several ones in the TCAM for insertion, the more the involved filters, the higher the chance of requiring more entry moves since more filters need to be rearranged (recall the CRG mentioned before). In addition, the number of layers is equal to the moves required in the worst case. Therefore, the performance may degrade as the number of layers increases.

The issue of where to allocate the empty entries is another factor which affects the performance of rearrangement. Most existing update schemes allocate a set of contiguous empty entries in the middle or end of the TCAM. To determine the position of empty entries, we first analyzed the distribution of filters based on layers. Table III shows the first three layers that include largest proportion of filters. The maximum numbers of layers are also shown in the fifth columns. Since there are more than 60% of the filters in layer 1, it's reasonable to conjecture that the new filter has greater chance to be inserted in layer 1. Hence, we allocate the empty entries

TABLE III. The first three layers that contain most filters

Filter Set	First	Second	Third	Maximum Layer
acl_10K	Layer 1 95.1%	Layer 2 3.49%	Layer 3 0.67%	28
fw_10K	Layer 1 60.5%	Layer 15 6.50%	Layer 18 4.66%	44
ipc_10K	Layer 1 79.2%	Layer 2 2.58%	Layer 23 1.34%	72

TABLE IV. Experimental Results for Entry Rearrangement

Filter Set	avg # of writes per insertion	avg # of writes per deletion
acl_10K	1.40	1.13
fw_10K	3.15	1.40
ipc_10K	5.24	1.39

between layer 1 and 2. The experimental results for entry rearrangement are shown in Table IV. Note that we use TCAM write operations to perform deletions by unsetting the valid bits of the deletion entries.

C. *Analysis with other schemes*

Although reference [1] does not need any TCAM entry moves to complete the update process, it still needs to update the priority graph first, and the priority values can be further updated. In addition, the search performance is worse than the other schemes. CoPTUA [2] requires much more time for moving the entries to maintain the consistent rule set, which delays the time for the new filter to take effect. PC-DUOS [3] uses two auxiliary data structures, so it takes more memory to maintain them. We also implemented the CoPTUA algorithm [2] to calculate the number of TCAM write operations required for update by using the same filter set and trace file. Since CoPTUA updates a batch of filters at the same time, we merged the trace files for insertion and deletion into one. Therefore, all the filters in update trace files are processed by CoPTUA simultaneously. The number of empty entries affects the efficiency of CoPTUA, so we provide three cases where 2%, 5%, 10%, and 15% of the filter set entries are empty in our simulation. As shown in Table V, at least 15 TCAM write operations are required for updating a filter in average although 15% of the filter set entries are empty. Note that the number of empty entries does not affect the performance of our configuration.

V. CONCLUSION

In this paper, we proposed an efficient algorithm for updating packet classifier. By utilizing the extra bits and capability of multiple matching, our scheme does not need to allocate a lot of memory to maintain an auxiliary data structure. Our experimental results show that only a few additional TCAM search cycles are needed to calculate the entry rearrangement. The numbers of TCAM entry moves are also small due to the approach of layer grouping and allocating the empty entries next to the layer that includes most filters.

TABLE V. Average number of TCAM write operations per update filter by using CoPTUA to update a batch of rules provided that 2%, 5%, 10%, and 15% of the filter set entries are empty.

Filter Set	2%	5%	10%	15%
acl_10K	23.56	22.61	16.24	15.65
fw_10K	28.72	24.26	21.19	20.05
ipc_10K	25.51	23.65	18.72	15.88

REFERENCES

[1] H. Song and J. Turner, “Fast Filter Updates for Packet Classification using TCAM”, GLOBECOM, 2006.

[2] Z. Wang, H. Che, M. Kumar, and S.K. Das, “CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking”,IEEE Transactions on Computers, 53, 12, pp.1602-1614. December 2004.

[3] T. Mishra and S.Sahni, “PC-DUOS: Fast TCAM Lookup and Update for Packet Classifiers”, ISCC - International Symposium on Computers and Communications, pp.265-270,2011.

[4] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," SIGCOMM’05, 2005.

[5] D. Shah and P. Gupta, “Fast Incremental Updates on Ternary-CAMs for Routing Lookups and Packet Classification”. In HotI, 2000.

[6] F. Yu, T.V. Lakshman, M. A. Motoyama, and R. H. Katz, “SSA: A power and memory efficient scheme to multi-match packet classification,” in Proc. Symp. Architect. Netw. Commun. Syst., Oct. 2005.

[7] D. E. Taylor and J. S. Turner, “ClassBench: A Packet Classification Benchmark”, IEEE/ACM Transactions on Networking, Volume 15, No. 3, pp.499-511,June 2007.

[8] F. Yu and R. H. Katz, "Efficient Multi-Match Packet Classification with TCAM”, Hot Interconnects, August, 2004.

[9] F. Zane, G. Narlikar, and A. Basu, “CoolCAMs: Power-efficient TCAMs for Forwarding Engines”, IEEE INFOCOM, 2003.

[10] A. L. Buchsbaum, G. S. Fowler, B. Krishnamurthy, K.-P. Vo, and J. Wang, "Fast Prefix Matching of Bounded Strings", ACM Journal of Experimental Algorithmics, vol. 8, Jan. 2003.

[11] T. Mishra and S.Sahni, “DUOS – Simple Dual TCAM architecture for routing tables with incremental update”, IEEE Symposium on Computers and Communications, 2010.

[12] R. Panigrahy and S. Sharma, "Reducing TCAM Power Consumption and Increasing Throughput," Proc. Hot Interconnects, 2001.

[13] K. Zheng, H. Che, Z. Wang, and B. Liu, "An Ultra High Throughput and Power Efficient TCAM based IP lookup Engine," Proc. IEEE INFOCOM, 2004.

[14] B. Vamanan and T.N. Vijaykumar, “TreeCAM: Decoupling Updates and Lookups in Packet Classification,” CoNEXT 2011.

[15] T. Banerjee-Mishra and S. Sahni, “Consistent Updates for Packet Classifiers,” IEEE Transactions on Computers 2012.

[16] J. Lunteren and T. Engbersen. Fast and Scalable Packet Classification. IEEE Journal on Selected Areas in Communications, 21, May 2003.

[17] Y.-K. Chang, C.-I. Lee, and C.-C. Su, “Multi-field range encoding for packet classification in TCAM,” in IEEE Infocom Mini-Conference, 2011.

[18] M. Adiletta, M.R. Bluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The Next Generation of Intel IXP Network Processors," Intel Technology J., vol. 6, no. 3, pp 6-18, 2002.